Automated Client/Server Based File Transfer Protocol: Optimized to Reduce Network Bandwidth and Provide File Prioritization

Benzil John, Michael Lewallen, Ramya Challa, Sultan Almutairi, Wade Moore

CSC 573 - Internet Protocols

Professor: Dr. Muhammad Shahzad

Teaching Assistants: Hassan Iqbal
Tae Hyun Kim

December 3rd, 2018

Component	Component weightage	Benzil John	Michael Lewallen	Ramya Challa	Sultan Almutairi	Wade Moore
High level design						
Algorithm development						
Coding						
Debugging						
Report Writing						
Totals	1.0	.18	.19	.19	.19	.25

Report Contents:

Abstract

This report outlines the motivation, design, implementation, and analysis of our team's Advanced File Transfer Protocol. The system is focused on two layers of the network stack: The Application and Transport layers. Through use of the HTTP Application layer protocol, and an implementation of our custom Transport layer UDP protocol, the system provides a solution for backing up file data from many clients to a single server. The following provides details about the design decisions made, and the limitations that had to be considered. In addition to evaluating the current product, this report provides guidance with how to extend and improve the implementation so to better accommodate commercial needs.

1. Introduction

With the increasing ability to utilize computer data storage, users are encouraged to redundantly store electronic data in multiple forms so to protect against loss. Data Backups are especially important for organizations whose many users are the employees or faculty who store data critical to the success of the organization. Upon further consideration, one realizes three fundamental requirements of the client-server backup system.

The first, one would consider consolidating this data into one or few central storage servers, to better manage the plethora of user data backups. Secondly, the process of syncing client data to the server must be automated so to not disrupt clients, and to ensure data is backed up accordingly. If the responsibility of initiating the sync process was reliant upon the client's invocation, the gap between data being updated and synchronization could be increased. Human forgetfulness, and the requirement of not disrupting clients, prevents this gap from being smaller on average than that of an automated system. This gap is of special concern being it is the time where if failure occurs, there would be data loss, and will be considered more later. Finally, the third requirement is to reduce the network bandwidth used to synchronize data.

As organizations increase the number of clients being managed, the cost of the backup system in respect to the use of the network can become significant. Many users backing up potentially large amounts of data could disrupt the performance of the business functions already dependent upon the network. This system shall implement intelligence that will allow only sending data which must be updated, neglecting data that is already accounted for. To recap, we assert that our data backup system *must* support a many-to-one (or few) client-server architecture, which can automatically detect and apply updates to the data while minimizing network impact.

In addition to the three fundamental requirements of our system, there is a practical need to provide support for data prioritization. As organizations grow, the amount of network bandwidth will inevitably increase accordingly. This introduces a greater latency to synchronize data, and there remains the "data loss gap" previously mentioned. Our system will support allowing clients to configure how small or large that gap is for different sets of their own data. Therefore, one's "critical_directory/" can be configured to be synced every five seconds, while another less important directory is configured to be synced every two minutes.

These four main requirements guided the design and implementation of our system. In doing so, we encountered several additional obstacles to overcome and decisions to make in respect to the design.

2. Design

The core design of our system can is intended to support variations to the computing platform or environment. However, higher priority features consumed the time needed to accommodate greater flexibility. Therefore, the following environment characteristics are included within this design specification:

- All clients and servers run Unix based operating systems. The only restriction *should* be that the clients and server share a consistent file system. However, there was not enough time to guarantee the Advanced FTP system functions properly on the Windows OS.
- All clients and server must be connected to an open network, which allows for insecure communication between clients & server.
- Sufficient information for clients to access the server is known beforehand.

If a computing device satisfies these specifications, either the client or server application may be run appropriately. Provided that the required software needed to run either application is installed; these are details left to the Implementation section.

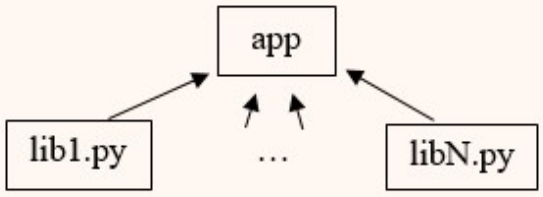
The client and server applications are two separate programs which are intended to run in the background on separate machines. The server is an API server which provides HTTP endpoints for incoming clients register and initiate sending data. The client is a running program which periodically checks the configured file directory for changes, and

if any are found, sends the data to the sever; otherwise, the client waits until it is time to check again. While very different processes, there is a purposeful degree of symmetry between the client and server designs resulting in many similarities. Both applications are designed to be invoked with a YAML[1] configuration file, and that is the full extent of user interaction with either program. Both the client and server applications run autonomously, communicating with each other to synchronize data based on the configuration settings. At any point the client or server may be stopped (via an interrupt signal) and then restarted the same way the application was started. Both the server and client applications store information allowing either to resume where it had left off. The exception being during data synchronization, if interrupted, the behavior is undefined. This is an area for improvement; however, if stopped when idle, each time either application is restarted the configuration is again read so updates may be applied. The various similarities between the client and server responsibilities led our team to develop each with the same model in mind. This model entails an "app" script which uses various library components defined in the "lib" to perform various functions. The app is the entry point of execution; however, most of the application logic resides in the library code. Figure 1 shows the source code layout for the server and client; followed by figure 2 which displays a simplified, generic class diagram for this approach.

Figure 1: Source Code Layout

server/	client/
|_app/	|_app/
|_ |_server	|_ |_client
|_lib/	|_lib/
|_ |_server_manager.py	|_ |_client_manager.py
|_ |_server_db.py	|_ |_client_db.py
|_ |_server_socket.py	|_ |_ client_socket.py
|_ |_recieving_queue.py	|_ |_ client_watcher.py
	|_ |_ sending_queue.py
	|_ |_file_renderer.py

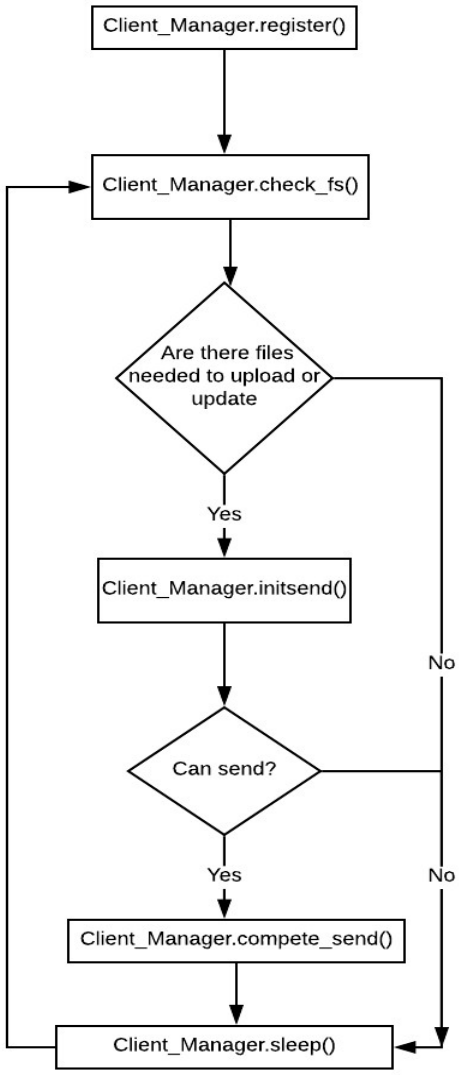
Figure 2: Generic Class Diagram



2.1 Client/Server Configuration

The configuration file of either application is critical because it contains information necessary for communication. Describing each configurable field provides insight into the design of various components, and how to use the system oneself.

Flowchart 1: Client side flow chart

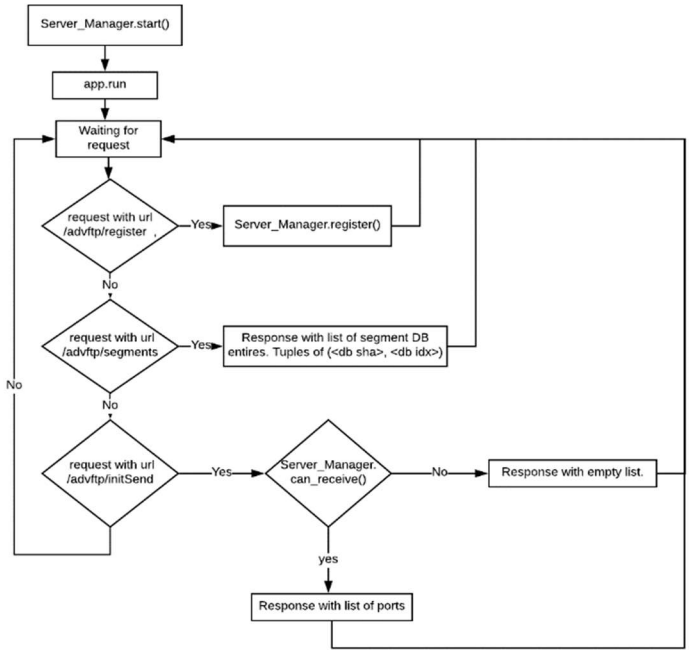


Client Fields:

- *server_http_url*: The address at which the server is hosted where the server application runs.
- server_socket_*address*: The host name of the client.
- *directories*: Will list all the paths to the directories which have to be synced.
  - *full_path*: Full path of the directory.

- *watch_interval*: Time in seconds period for which the directory is checked for any modifications in the files.
- *num_send_threads_per_file*: The number of threads that send the segments to the server in parallel.

Flowchart 2: Server side flowchart



Server Fields:

- *client_data_dir*: Path to the directory in server where the client files are stored.
- *segment_size*: The number of segments the file is divided into.
- *max_num_ports_per_host*: Maximum number of UDP connections per client.

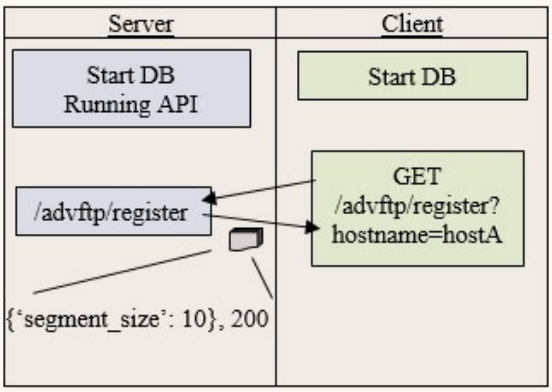
2.2 Client/Server Communication: Registration

To further describe the design of our advanced FTP system, we describe the exact communication pattern between an arbitrary client and the server. To begin, the following is an outline of the initial communication pattern; when the client first initiates communication with an existing server.

At this point we assume there is a running server reachable through a common network. Once started, the server determines if there is an existing database file, creating one if not found. From there the server merely waits for client connections. The client will initiate communication with the server over HTTP using the base URL set in the configuration as defined above. Note, the following discussions regarding any of the server's API endpoints will be relative to this base URL. With these assumptions we can focus on the client host which has the correct configuration for connecting to the server. One invoked, the client will also check if there is an existing database file, also creating one if not found. However, now the client immediately attempts to register with the server by sending a GET HTTP request to the server endpoint: "***/advftp/register?hostname=<client-hostname>***". One may notice that the client hostname is provided to server, this will distinguish this client from others already registered with the server. The server will allocate a new directory for the registering host, named based on the convention: "*<client-hostname>_dir*". Afterwards the server will add the new client to its database, then respond to the client confirming successful registration and provide the current segment size. As mentioned in the configuration definition, the segment size effects how network bandwidth minimized by only sending updated segments of data rather than the entire file. This parameter is critical to cooperative communication between the client and server. Therefore, to ensure its consistency, the server will provide this value to the client directly at the start of each session. The client will start a session by registering each time the program is started. If the client has already registered then the server need not reallocate anything, but merely return the current segment size.

Once the file has received and stored the current segment size, the client can begin the process of checking the configured file directories. Initially, we can imagine any existing files will all have to be sent in their entirety. This would be the heaviest load and should be considered when starting clients. An improvement that could be made to the client application could involve limiting how many files can be sent at once. However, part of our reasoning not to implement such logic was because clients would likely desire unsent files to be uploaded as soon as possible to minimize the loss gap. The communication pattern for sending data between the client and server is described in detail further below. Directly below is figure 3, a flow-chart representation of the client-server communication pattern during registration.

Figure 3: Client Registration



2.3 Client/Server Communication: Data Synchronization

Once the client has registered with the server, the client and server can at any point appropriately loop through the following steps:

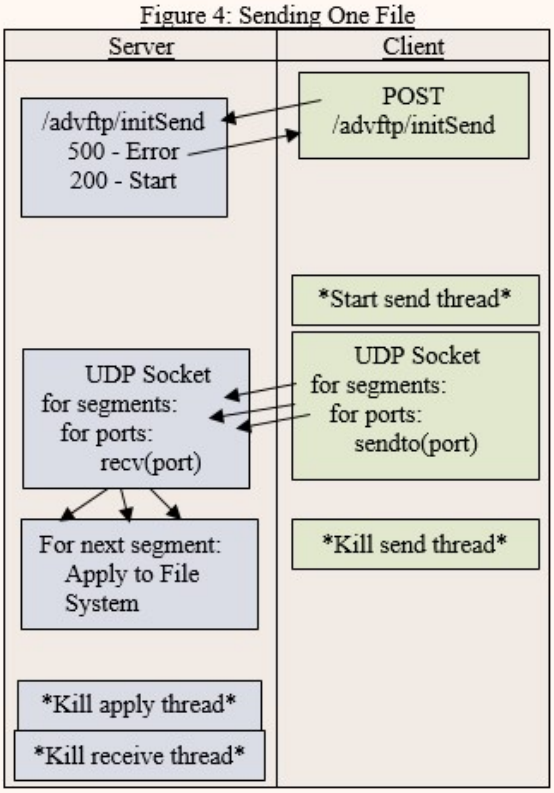
1. As mentioned in the configuration definition, the items of interest are the configured directories, their intervals, and current counters. Each iteration will sleep until the next one or more directories must be checked. Upon then, the client will determine which files have been updated based on the last modified timestamp stored in the client database. Update each file in the database as needed and return the list of file paths which have been updated or added.
2. The client uses another HTTP API endpoint of the server to get the list of segments for each file. ***'/advftp/segments?hostname=<H>&filename=<F>'***
   If the file has not yet been added to the server then an empty list is returned. Otherwise, the list of SHAs for the file is returned to the client in order by index.
3. For each file in the returned list, read the file in chunks of segment sized bytes. Each segment will include its SHA256 checksum, the index within the file, and the filename. The filename is relative to its configured directory; meaning the base path of the file, which is host specific, is omitted. If the server returned a non-empty list of file SHAs, then compare each one of the corresponding index to that being currently calculated. If they match, then the current segment can be omitted. Again, both the SHA of the content, and the index number must match to be omitted. **This is how network bandwidth is reduced.** Large files will take considerable time to upload initially (if not made large gradually), but subsequent edits may only change small parts of the large file. This implementation will exploit this and only send those few segments that were updated, as opposed to the entire file.
   {
   'content': <decoded segment-sized bytes>,
   'sha': <SHA256 value of content>,
   'filename': <corresponding relative filename>,
   'index': <segment index in file>
   }
4. Now the client has the list of filtered segments which need to be sent to the server. Add all of these to the *sending queue* and then start a sending procedure.
5. The start of a sending procedure entails sending another HTTP request to the server, this time a PUT request. The following is provided in the form of the request:
   * hostname
   * filename
   * number of segments
   * segment size

   All of which are required to effectively handle transferring all segments. While receiving UDP packets from the client, specified by hostname, the maximum size (in bytes) of each incoming packet is required. The segment size provided here initially is really the total size of the entire segment object created. This includes the content which is of "segment size", a SHA value, index, and filename attributes. This is a consistent size for each file, except potentially the last segment of a file, but the upper bound accommodates this well. Notably, all segments would have equal sizes if not for the filename, which could vary in size. The filename is required so the server can apply new segments as they arrive and have enough information to work autonomously.
6. The server accepts the PUT request from a client and then determines if the number of currently open connections is greater than a configured threshold. If so then the client is refused from being able to send, indicated by the appropriate return code. The client will wait and repeatedly try to start the sending procedure with the server. Each time the current load is evaluated. However, if the load is not too high, then the server will allocate and bind to the next available ports in its configured range (By default: 10000+). The number of ports allocated for each file largely determines how quickly the file may be sent. This maximum is configured by the sever and is automatically decreased if the load is high. The list of one or more ports is returned to the server as confirmation that the sending procedure has begun.

7. The server immediately starts two threads for each file transfer with a host. One thread continuously reads new segments from the client over UDP sockets on the previously allocated ports. As each segment is decoded, it is added to the *receiving queue*. The second thread continuously checks for new segments to be added to the queue and then pops the next segment, so it can be applied to the server's file system. Each segment contains the content of the file to write, the relative filename, and the index; sufficient for writing the file. The full path of the file is stored in the sever database, and the offset of the content is determined by the (*segment_size*index*).
8. Once all of the segments have been sent by the client, the sending thread is killed. Once all of the segments have been read by the server, the server receive thread waits for those remaining file segments to be applied.
9. The server's apply thread doesn't know there are no more segments coming, it merely waits for more. The apply thread will kill this thread, deallocate and unbind the file's ports, and exit. This concludes the file transfer.

To recap, the client will for each file determine which segments to send, then start a separate thread to send them all. The server will accept a client's request to send by creating two separate threads. One will read new segments, and the other will apply them to the file system. Upon each iteration of sending segments, the client will send one segment on each of the ports provided by the server. Figure four below provides a flow-chart diagram to depict this communication pattern.

Figure 4: Sending One File



3. Implementation

The implementation of our Advanced FTP system is nearly aligned with the original design. However, there did arise considerations which forced us to reconsider the system functionality. Before discussing those considerations, the required technological specifications are outlined below.

3.1 Technology and Demo Script

The entirety of the Advanced FTP software solution is written in Python3, using various pip3 packages. The list of all third-party packages used without implementation is kept in the file: **"install_packages.sh".** Beyond this, there is no other further installation requirements to run our software. To see a simple and clean example of our system in use, the Makefile provided within the project source repository can be used. A default configuration for both the client and server is provided with the source under the "etc/" directory. Therefore, all that is needed to start the entire system is the following command: '**make demo**'. This will create four separate terminals. (1) The server process which waits for client connections; (2) The client process which checks the default directory and sends updates. (3) A simple process which repeatedly refreshes the contents of the Performance DB, and (4) a background process which edits the file system

under the default client directory. Provided this setup, a server can establish itself and accept file changes from a single client over the local network. This of course defeats the purpose of our system, which should backup files by placing them on separate hardware. However, this demo script provides an easy and consistent way to help test, evaluate, and develop our system. Further inspection of the provided Makefile should reveal targets related to running specific components (i.e. "**make run-server"**).

3.2 UDP versus TCP

During the development of our system, the most significant paradigm shift that needed to be made was the decision to implement the socket protocol using UDP instead of TCP. This was because during the process of sending many segments over TCP, there was the issue of distinguishing between one segment and the next. The underlying client operating system would group multiple packets that has been sent by the client, so to optimize the actually network traffic of the client. However, there is not a reasonable way to handle receiving potentially more than one segments. This was a great example of how TCP is a "byte-stream" instead a "message-based" protocol. Therefore, to easily ensure that segments are sent as separate message, the UDP protocol was used. This also provided better performance for obvious reasons. However, the major fault of using UDP is that the segments are now no longer guaranteed to be send. Our system now requires a mechanism of having the server retransmit dropped segments. The server already has enough information to provide this feature; the total number of segments for each file is provided to the server before starting the sending procedure. Furthermore, the index number of each segment in the file is also included; therefore, we could evaluate which indices were not received after waiting for a configured amount of time. However, while this implementation was certainly feasible, there was not enough time to implement this logic provided the decision to use UDP was not anticipated.

3.3 High Performance Mutli-Threaded Architecture

The current implementation of the Advanced FTP system does support various levels of parallelism within both the server and client. The client of course determines a list of files and corresponding segments which need to be sent. After adding all of the segments to the sending queue it is a matter of sending each one to the sever. The client will create X additional threads where each will simply loop over the sending queue either removing and sending the next segment or waiting for more to be added. The number of threads created for each file being sent by the client, (X), is configured within the client configuration file under field: "***num_send_threads_per_file".*** Once all segments have completed sending, then the main thread of the client will forcibly kill all currently running sending threads.

The server on the other hand has two independent jobs which are both needed for receiving a file from a client. The first a separate thread for receiving all segments from a client; decoding the data, updating the file segments database, and adding the new segment to a queue to be applied to the file system. The other thread is the one which applies incoming segments to the local file system. As clients initiate sending a single file to the server, the server will create two threads each time to handle the transaction. Appropriately joining or killing the running threads as each file transaction competes.

The final form of parallelism which was implemented within our system was in respect to the number of ports allocated to the client for sending a single file. The simplest scenario involves setting this configuration value to one; here the server will provide the client a single port to connect over once a file transaction is initiated. In this way, the client iterates through the remaining segments, each time sending one over the provided port. However, if the server is configured to allow multiple ports per file, then the sever could provide a list of port to the client when initiating a file transaction (e.g. [10000, 10001, 10002]). Now when the client iterates over the remaining segments, each time the client will send a single segment over each port. Likewise, the server will attempt to read a segment from each of the allocated ports, and then adds all of those to the receiving queue. One should note, when running the system with multiple send threads being created for each file, the number of ports for each file should also be more than one. Otherwise, the lower of the two values is the bottleneck.

3.4 Client Directory Prioritization

The originally proposal focused on the ability to allow clients to prioritize their files. This prioritization will allow certain directories to be checked more frequently than others, which will ensure updates are uploaded to the server quickly. This feature can be leverage easily through the client configuration file, specifically the "directories" section. The list of directories specified each include an "interval" which denotes the time in seconds between checking that directory for updates. The client will know to sleep until the next directory is ready to be checked.

This specification allows clients to organize their file data in a manner that may represent the following: One directory which is of low priority, so the interval is set to 300 (five minutes). And then another directory which contains production artifacts can be set to an interval of 5 (five seconds). We can image there is a much less likely chance of losing data in the production directory as opposed to the former.

4. Results and Discussion

The performance of the Advanced FTP was tested using a variety of parameters. First, three test files that contained 0.1KB, 1KB, and 10KB of data were created. Initially all three files are uploaded simultaneously to the server. Then the large 10KB file was modified by adding data at the end of the file and modified a second time by adding data at the beginning of the file. These five file uploads have corresponding round trip times in the performance database table. This provides five data points creating the average time calculated for each scenario:

1. 0.1 KB Initial send
2. 1 KB Initial send
3. 10 KB Initial send
4. 10 KB Edit at beginning of file
5. 10 KB Edit at end of file

Additionally, the files were uploaded under a variety of conditions to better understand how modifying the segment size, number of ports, and the number of threads impacted the performance of advanced FTP. As expected, it was observed that less time was required to upload the smaller data files as compared to the larger files. Furthermore, the time to edit the file after already being uploaded was tremendously faster; revealing the benefit of the projects design. However, the performance of advanced FTP did vary under different conditions. A summary of the average amount time that was needed to upload the files is shown in the graph below.

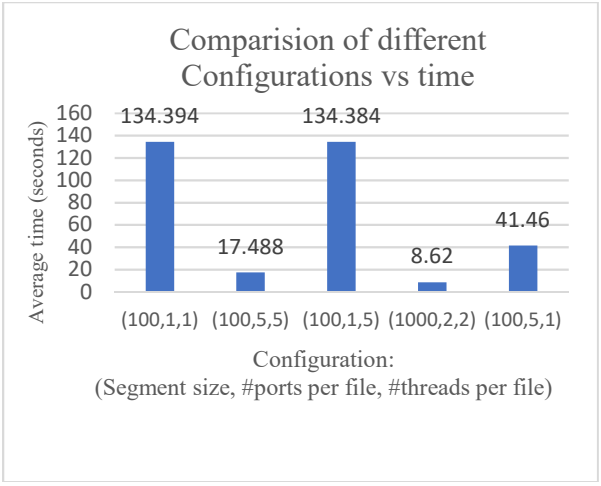
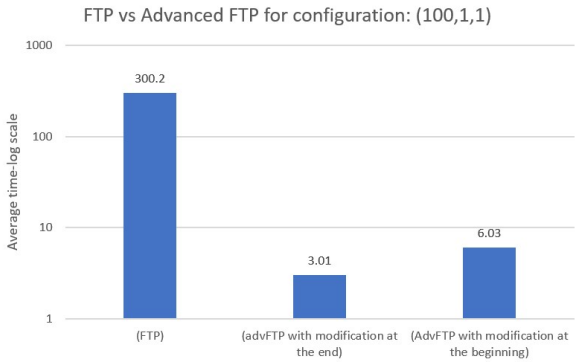


Figure 5: Average Upload Times

Based on these test scenarios, several key observations can be made. First, it is clear the total upload time is greatly reduced when a larger number of ports are used. Second, the upload time can be further reduced by increasing the segment size that is used as shown in the fourth case. Finally, we predicted that the benefit of multiple ports per file, or multiple sending threads per file, is negligible if one or the other is significantly smaller (or one). This was seen in the results, and more interestingly, when there are many sending threads, but few ports per file, the performance decreased compared to the serial version.

While performing these test scenarios, a few additional observations were made regarding how modifying the large 10KB file impacted the performance of advanced FTP. In each case, it was observed that more time was needed to upload changes that were made at the beginning of a file as compared to changes made at the end of the file. The graph below shows the differences in the amount of time needed to upload the different versions of the large 10KB file from one of the test scenarios.



5. Conclusions

As we observed in the results section, on an average, advance FTP achieves a speedup of 6000 over the traditional FTP. By sending smaller segments to the server, bandwidth is conserved. We also proved that, by increasing the number of ports and the number of threads for sending the segments will significantly increase the performance. Future scope for the project would be 1. To make the system more scalable, as the increase in number of clients will require the server to have more ports in order to maintain the connections. 2. To include a packet loss recovery mechanism as our protocol uses UDP as the transport layer protocol.

6. References

[1] James F. Kurose, Keith W. Ross. Computer Networking A Top-Down Approach Featuring the internet. ISBN 0-201-47711-4 Addison Wesley Longman Inc.

[2] https://docs.python.org/3/

[3] Nitin Agarwal, Akshat Aranya, Cristiam Ungureanu, “Reliable, Consistent, and Efficient Data Sync for Mobile Apps”. ISBN 978-1-931971-201

[4] Y. Cui, Z. Lai and N. Dai, "A first look at mobile cloud storage services: architecture, experimentation, and challenges," in IEEE Network, vol. 30, no. 4, pp. 16-21, July-August 2016. doi: 10.1109/MNET.2016.7513859